The Anatomy of Software Changes and Bugs in Autonomous Operating System

Katerina Goseva-Popstojanova
Lane Dept. Computer Science & Electrical Engineering
West Virginia University
Morgantown, WV 26506, USA
Katerina.Goseva@mail.wvu.edu

Denny Hood
Lane Dept. Computer Science & Electrical Engineering
West Virginia University
Morgantown, WV 26506, USA
dlh0038@mix.wvu.edu

Johann Schumann
KBR/WYLE, NASA Ames Research Center
Moffett Field, CA 94035, USA
johann.m.schumann@nasa.gov

Noble Nkwocha
Katherine Johnson NASA IV&V Facility
Fairmont, WV 26554, USA
noble.n.nkwocha@nasa.gov

***Abstract*—Cyberphysical systems with autonomous functions are complex pieces of software, consisting of many components, some of which implement autonomous functionality and some may use AI or machine learning algorithms. Software bugs in an autonomous system are of particular concern, as they can have catastrophic consequences. However, detailed studies based on empirical data are rare and therefore these bugs are not well understood. This paper aims to contribute towards filling that gap by investigating the software changes and bugs in Autonomy Operating System (AOS) for Unmanned Aircraft Systems (UAS), which consist of 26 components containing about 103,000 lines of code and having a total of 772 bugfixes. Based on the data extracted from the code repository and semi-structured interviews with the developers of AOS, we explore the differences among autonomous software components, components developed using Model-based Software Engineering, and reuse with respect to change proneness, fault proneness, distribution of bugfixes among AOS components and files of these components, and characteristics of bugs of different AOS components. Our results show that the autonomous components were significantly more change prone (measured in number of commits and code churn) and fault prone (measured in bugfixes per KLoC) than non-autonomous components. The distribution of the locations of bugfixes was skewed, both at component and file level (i.e., a small number of components / files contained the majority of bugs). These evidence-based findings provide important insights to researchers and practitioners alike and can be used to efficiently improve the quality and reliability of autonomous systems.**

***Index Terms*—autonomous operating system, unmanned aircraft systems, change proneness, fault proneness, empirical analysis, qualitative analysis.**

I. Introduction

Cyberphysical systems with autonomous functions are becoming more and more prominent in many important application areas, ranging from robotics and self-driving cars to autonomous aircraft and space probes. The software of such systems consists of many components and subsystems; specific autonomous components implement the autonomous functionality of the entire system. Often such autonomous components use Artificial Intelligence (AI) and machine learning (ML) algorithms. In the literature and general discussion, these autonomous components are the components everybody talks about. However, is that all?

A complex cyberphysical system has numerous software components, ranging from low-level drivers, middleware, controllers, to sensor-processing, perception, and the autonomous components that were mentioned before. Although the number of autonomous components in a system might be small compared to the overall architecture, the autonomous components must smoothly work and communicate with the rest of the system. Potentially other, non-autonomous components must be modified to work together with autonomous components. In particular for safety-critical applications, like autonomous drones or aircrafts, a perfect interaction between autonomous components and non-autonomous components is mandatory for safe system operation. These are crucial issues for any certification attempt.

Software bugs in autonomous systems can lead to system crashes, hangs, and undefined behaviors, resulting in catastrophic consequences. Therefore, understanding software bugs in autonomous systems is crucial for ensuring their reliability, robustness, safety, and security. Many empirical studies focused on software bugs exist, for a variety of domains including open-source software (e.g., [1], [2]), space (e.g., [1], [3], [4]), numerical software (e.g., [5]), machine learning libraries (e.g., [6], [7]), and software developed using model-based software engineering and incorporating autogenerated code [8]. However, software bugs of autonomous systems have not been studied extensively and therefore currently are not well understood. The limited current knowledge is based on only several recent works that were focused on empirical analysis of software bugs in autonomous systems [9]–[12].

To fill this gap, we conducted a detailed empirical study of software changes and bugs of a complex autonomous system – the AOS (Autonomy Operating System) – which is a flight software used to support autonomous operations of an Unmanned Aircraft System (UAS) in the National Airspace. The AOS [13]–[15] was developed at the NASA

Ames Research Center and successfully test flown on a highly customized 20 lbs drone. Specifically, we analyze 26 AOS components, which are referred to as apps in the context of the AOS project. (In this paper, we use "component" and "app" in a synonymous way.) Out of 26 apps, eight had autonomous functionality. Six of these were developed using Model Based Software Engineering (MBSwE). Eight apps included reused software; two of these had autonomous functionality and were developed using MBSwE.

In addition to the sizes of AOS apps (measured in lines of code (LoC)), we study their change proneness and fault proneness. Specifically, we measure the *change proneness* of each app in terms of (i) *the number of commits* made to the version control repository and (ii) *code churn* defined as the sum of LoC added and LoC removed for all commits for a given app. Following prior work (e.g., see [22] and references therein), a commit that involved fixing bugs was labeled as a *bugfix*. For the analysis at app level, if a commit that included fixing bugs led to changes in only one app, even if several files in that app were changed, we counted it as one bugfix at app level. If a commit with bugfixes led to changes in multiple apps, we counted it as one bugfix for each app. The *fault proneness* of each app was captured by the number of bugfixes associated with that app. To account for vastly different sizes of AOS apps, we also explored the code churn and number of bugfixes normalized by size (i.e., code churn (in LoC) per KLoC and number of bugfixes per KLoC).

The empirical analysis presented in this paper is based on data extracted from the Git version control repository, which stored all artifacts and the version history of the AOS through its entire design and development process. (Details on data extraction and metrics are given in Section V.) For additional in-depth information, the authors interviewed several developers and researchers of the AOS project. This approach allowed us to address the following research questions in this paper:

RQ1: What are the typical sizes of individual AOS software apps?

RQ2: Are there differences in change proneness (measured in number of commits and code churn) among apps from different categories (i.e., Autonomous, MBSwE, Reused)?

RQ3: Are there differences in fault proneness among apps from different categories (i.e., Autonomous, MBSwE, Reused)?

RQ4: Are the locations of bugfixes unevenly distributed among the apps' files?

RQ5: What can be learned about the characteristics of software bugs from qualitative analysis?

The main contributions of this paper include:

- We analyzed a large, complex autonomous software system consisting of 26 components containing about 103,000 lines of code in over 980 files. In particular, we explored the change proneness, fault proneness, spread of bugfixes at component and file level, and characteristics of bugfixes of different components. None of the related works on empirical analysis of bugs in autonomous systems [9]–[12] explored the change proneness, while two works [9], [11] investigated the spread of the bugs but only at component level. We identified a total of 772 bugfixes in the 26 AOS components. The results of our analysis showed that the majority of bugfixes were located in a small number of AOS components, and within components in a small number of files. These findings are consistent with the Pareto principle of uneven distribution of bugfixes across files observed in empirical studies in other domains (e.g., [1], [23]) and indicate that allocating more development and testing effort to these parts of the software provides cost effective way to improve its reliability, safety, and security.
- We specifically analyzed the differences of change proneness and fault proneness among autonomous software components, components developed using MBSwE, and reuse. Related works [9]–[12] analyzed only the bugs in the autonomous parts of the systems under consideration, and did not explore the distinction with non-autonomous components, and with components developed using MBSwE or reuse. Our work, for the first time, provides empirical evidence that the autonomous components are significantly more prone to changes and bugs compared to non-autonomous components, which indicates the challenges associated with the development of autonomous functionality.
- We conducted detailed qualitative analysis of software bugs for individual components on the component and file level. Their skewed distribution might provide insights to avoid similar bugs in future autonomous systems.

The rest of this paper is structured as follows. After discussing the related works in Section II, we briefly present the architecture and autonomous functionality of the AOS system as background in Section III. Section IV presents the AOS development process and basic AOS facts. In Section V describes our data extraction methods and metrics used in this study. Section VI presents the empirical results of the study, regarding the size of each app, code changes, and quantitative and qualitative analysis of bugfixes. Section VII discusses the threats to validity, and Section VIII summarizes the major findings and provides concluding remarks.

II. Related Work

Empirical analysis of software bugs has been a very active research area, with many studies focused on a variety of domains including open-source software (e.g., [1], [2]), space (e.g., [1], [3], [4]), numerical software (e.g., [5]), machine learning libraries (e.g., [6], [7]), and software developed using model-based software engineering and incorporating auto-generated code [8]. However, software bugs of autonomous systems have not been studied, with an exception of several recent papers described in this section.

Two recent works were focused on studying autonomous vehicle (AV) bugs [9], [10]. Garcia et al. [9] presented an empirical study of bugs of Apollo and Autowarein, two open source Autonomous Driving Systems (ADS) that operate at autonomy level 4 (L4) which is the autonomy level that does not require human drivers to stay alert and ready to take over control[1]. The authors extracted 499 AV bugs from the commit logs of the Apollo and Autoware repositories in GitHub and using manual analysis classified the root causes and symptoms of the bugs, as well as identified the AV components affected by these bugs. Specifically, that work used 13 root causes (algorithm, numerical computation, assignment, missing condition checks, data, misuse of an external interface, misuse of an internal interface, condition logic, concurrency, memory, documentation, configuration, and other) and 20 symptoms (e.g., crashes, hangs, build, launched, IO, obstacle processing, logic). The AV components with significant number of software bugs were: CAN Bus, Control, HD Map, Localization, Perception, Planning, and Prediction. The most fault prone component Planning had significantly more bugs than the other components (i.e., 27.05% of all bugs in both systems) followed by Perception component which had 16.63% of all AV bugs.

Another study focused on analysis of AV bugs was conducted by Tang et al. [10] and based on OpenPilot, which is an open-source driver assistant system that belongs to level 2 (L2) autonomy and supports functions such as adaptive cruise control, automated lane centering, forward collision warning, etc. In total, the authors collected 235 bugs from the pull requests and issues of the OpenPilot project. These bugs were classified into five categories: model bugs, plan/control bugs, car bugs, hardware bugs, and UI bugs. The results showed that the car bugs related to the interface with different car models dominated with 31.48%. Plan/control bugs were ranked second highest ranked with 25.95%.

The bugs in UAS were studied in two recent papers [11], [12], both based on two open source software suites PX4 (capable of controlling drones) and ArduPilot (capable of controlling unmanned vehicle systems such as drones, fixed-wing and VTOL aircrafts, helicopters, ground rovers, boats, submarines, and antenna trackers). Taylor et al. [11] extracted 277 firmware bugs in the ArduPilot and PX4 code bases and investigated the root causes, symptoms, reproducability of the bugs, and location at component level. They used 5 root causes (i.e., semantic, sensor, memory, concurrency, and other) and found that the semantic bugs dominated, accounting for 65% of all bugs, followed by incorrect use of sensor values which occurred in 22% of bugs. This work used 5 symptoms and the analysis showed that over 21% of bugs were dangerous, that is, their symptoms included crashes or FlyAways (i.e., the pilot loses control). With respect to reproducability of UAS firmware bugs, the analysis showed that 74% of bugs were reproducible under default settings, 20% were reproducible under modified settings, and 6% were reproducible only under specific hardware or timing conditions. The distribution of bugs was studied at component level and it was concluded that out of five components HAL had the most bugs (i.e., 29%) followed by State Estimation with 19% of the bugs.

Wang et al. [12] also conducted an empirical study based on ArduPilot and PX4. They extracted 569 bugs from these two projects, using three sources of data, i.e., bug reports, commits, and bug patches on GitHub. Following an iterative manual labeling process, out of the 569 bugs, 168 were identified as UAS-specific bugs and their root causes were classified in 8 categories (i.e., limit, math, inconsistency, priority, parameter, hardware support, correction, and initialization). In addition, this paper discussed the challenges in detecting and fixing the UAS bugs and how to address them.

Some works have investigated software bugs in autonomous systems with a different goal than empirical characterization. For example, Timperley et al. [17] extracted 228 bugs from the commit logs of the ArduPilot project's version-control history and studied the bugs from the perspective of reproducibility in simulation used for testing. Others had broader scope and studied the software bugs in cyberphysical systems using data collected from 14 open-source projects [18].

It appears that none of the related works on empirical analysis of bugs in autonomous systems [9]–[12] explored the change proneness. Two works [9], [11] investigated the spread of the bugs but only at a coarse level of system components. Furthermore, related works [9]–[12] analyzed only the bugs in the autonomous parts of the systems under consideration, and did not explore the distinction with non-autonomous components, and with components developed using MBSwE or reuse (if any).

III. AOS Background: Architecture and Autonomous Functionality

AOS is a software system that enables core capabilities for the autonomous operations of an unmanned aircraft [13]. It is based on NASA's Core Flight System (cFS) [19], which provides a high-level layers of infrastructure and communication mechanisms.

The AOS architecture, shown in Fig. 1 provides capabilities for the execution of flight plans, natural-language communication with Air Traffic Control [14], Diagnostics, Prognostics, and contingency planning [15]. The underlying cFS system provides a "software bus," which is a publish-subscribe architecture that is used for communication between the different components or applications (called "apps") of the system.The apps, which implement and support autonomous operation are shown as yellow boxes. They use different languages, algorithms, and tools for their task, and they form a knowledge-bus and autonomy executive layer (shown in red at the top). The apps on the left-hand side of the figure show autonomous functionality, like specific operational procedures and contingency management. The entire AOS contains 43 applications, many of which are part of the cFS/cFE distribution.

Fig. 1 shows the AOS architecture from a component point of view. Each of the components in the AOS box are individual cFS apps that can be activated on a regular schedule, in case of AOS with a rate of up to 10Hz. AOS is communicating with

[1]The Society of Automotive Engineers (SAE) [16] defined six levels of vehicle autonomy, with Level 0 (L0) being the lowest (i.e., no autonomy) and Level 5 (L5) being the highest (i.e., full autonomy in any driving environment).

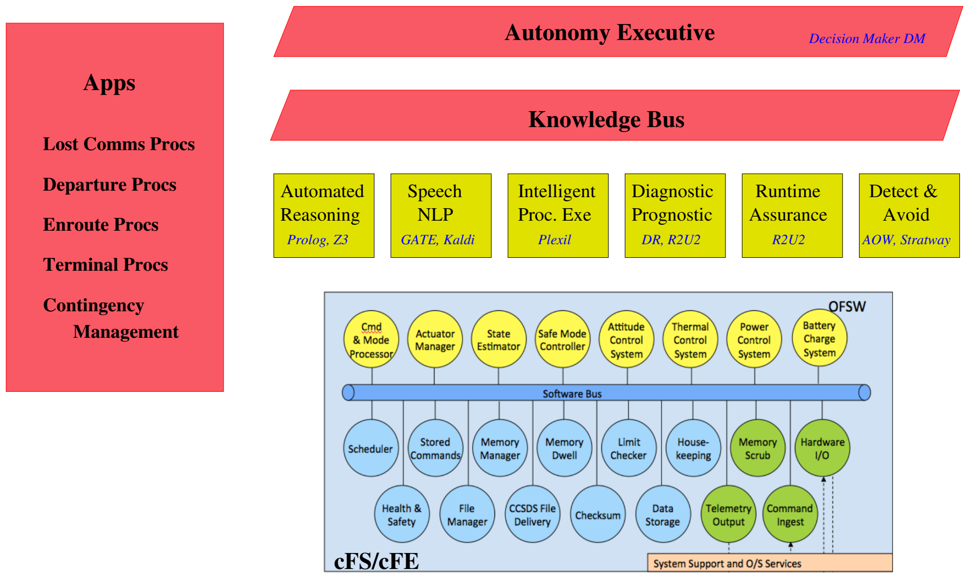

Fig. 1. The AOS system shown in an "app"-centric view. cFS/cFE provides the basic infrastructure for autonomy operations.

a low level flight control software (Fig. 1 bottom) to obtain sensor and aircraft status information and to issue low level commands. Specifically, a slightly modified version of the open-source ArduPilot software (http://ardupilot.org/copter/) is used, running on a PixHawk PX4 hardware (pixhawk.org), which directly interfaces with sensors and controls the motors of the UAS and is in charge of keeping the aircraft in the air and performing waypoint-to-waypoint flights.

AOS's model-based diagnostics, prognostics, and contingency planning framework is implemented as several apps on top of the cFS system and centered around the Decision Maker (DM). Fig. 1 shows the flow of information: sensor and status data from the UAS are transmitted by the ArduCopter flight software into the AOS system. These data are used to perform diagnosis, prognostics, and monitoring (Fig. 1, components shown on the left). The Diagnostic Reasoner (DR) performs model-based detection and isolation of failures, using an efficient algorithm based upon diagnosability matrices [15]. The R2U2 (Realizable, Responsive, Unobtrusive Unit) is continuously monitoring the system behavior using temporal logic observers and Bayesian reasoners. The on-board prognostics (oPRG) engine performs model-based determination of the state-of-charge (SOC) and rest of useful life (RUL) for the flight battery. More elaborate prognostics tasks that would consume too much on-board resources are handled by the PaaS (Prognostics as a Service) client, which uses a ground-based server for the computation.

AOS is separating the capability of health and contingency management into two modules: diagnosis/prognostics and decision-making. In the case of an adverse event or failure, diagnosis is performed first, to determine its root cause. Then, the UAS's current health status is handed over to the Decision Maker (DM) component.

The DM performs logic-based search to find (i) active diagnostic procedures to improve the diagnostic resolution (if necessary) and (ii) contingency flight plans, which can be safely executed under the current circumstances. If necessary, such contingency plans might contain emergency actions like, for example, cutting short the flight, diversion to a nearby airport for emergency landing, or an immediate ditch by activating an on-board parachute. This functionality allows the UAS to perform autonomous operations even under adverse conditions. The generated active diagnosis or contingency plan is sent to Plexil [20]. Plexil is an event-driven planner that has been customized to execute flight plans for nominal/off-nominal operations, which might require Air Traffic Control (ATC) interaction. Spoken ATC commands are processed by the Natural Language Processing (NLP) unit in AOS [14]. During plan execution, Plexil emits a sequence of commands and waypoints to the low-level autopilot that the UAS will follow.

IV. AOS Development Process and Basic Facts

The AOS software was developed as a research effort; thus its development did not have formal releases. The major milestones of the project corresponded to important test flights that have been carried out using a DJI S1000+ octocopter.

There are a total of 137 development branches in the AOS development repository. Of these branches, we selected a specific branch which was used for a major test flight milestone.

Our analysis is based on 26 AOS apps. 17 additional apps were excluded from the analysis either because they were very small (e.g., just one file) or because of the lack of use. Table I shows the essential details for the 26 apps that were analyzed in this study. We use the following notation: *A* for an app realizing an autonomy function, *M* for an app that was developed using MBSwE ans/or automated code generation,

TABLE I
BASIC FACTS OF AOS APPS WITH CATEGORY AND PROGRAMMING LANGUAGES (% OF FILES PER APP; ONLY VALUES > 10% SHOWN). PLP, PLE, AND PLX FILES CONTAIN DOMAIN-SPECIFIC PLEXIL PLANNING LANGUAGE

App	Description	Category	LoC	#Files	Languages
nlpio	Natural Language Processing I/O	A	1,896	15	h (65%), c (25%), txt, sh
prolog	Logic-based navigation	A	2,228	53	prolog (58%), h (19%), c (15%), pl, txt, sh
DM	Decision Maker	A, M	3,879	54	prolog (74%), h (17%), c, txt
plex_mav	Plexil to MAV (Micro Air Vehicle) interface	A, M	848	8	h (74%), c (13%), txt (13%)
plex_nlp	Plexil Nat. Language processing interface	A, M	490	8	h (74%), c (13%), txt (13%)
R2U2 w/AGC	Runtime monitoring with AGC	A, M	29,284	129	c (49%), h (48%), cc, txt
R2U2	Runtime monitoring	A, M	3,494	51	c (51%), h (43%), cc, txt
lc	Limit checker	A, M, R	2,013	8	c (49%), h (43%), txt (13%)
plexil	Plexil Planner	A, M, R	28,778	439	plp (27%), ple (25%), h (13%), c, hh, cc, plx, txt, sh
cfs_lib	Core Flight System Library	R	284	2	h (50%), c (50%)
ds	cFS Data Storage	R	1,832	6	c (50%), h (33%), txt (17%)
hk	cFS House Keeping	R	775	7	h (57%), c (29%), txt (14%)
mm	cFS Memory Manager	R	2,879	11	c (64%), h (36%)
sch	cFS Scheduler	R	1,171	5	h (40%), c (40%), txt (20%)
ttsio	text-to-speech app	R	441	10	h (67%), sh (17%), c, txt
aos_lib	AOS utility library		720	16	h (35%), c (32%), cc (14%), hh (14%), txt
cf	Core Flight System		8,587	18	c (77%), h (17%), txt
cs	Checksum		2,419	8	h (49%), c (38%), txt (13%)
DR	Diagnostic Reasoner		2,396	30	h (56%), c (37%), txt
fm	cFS File Manager		1,803	6	c (50%), h (33%), txt (17%)
hs	cFS Health and Safety		2,352	21	c (62%), h (35%), txt
ib	Instrumentation bridge		308	10	h (70%), c (20%), txt (10%)
mav_bridge	cFS to MAV interface		1,704	20	h (90%), c, txt
md	cFS Memory Dwell		1,154	6	c (50%), h (33%), txt (17%)
sc	cFS Stored Command		4,490	25	c (88%), h, c
si	cFS Stream Input		648	9	h (67%), c (22%), txt (11%)
simple	SIL testing without autopilot		439	8	h (74%), c (13%), txt (13%)
Total			103,818	983	

and *R* for an app where major parts of the code have been reused. Out of the 26 apps, eight had autonomous functionality. Six of these were developed using Model Based Software Engineering (MBSwE). Eight apps included reused software; two of these had autonomous functionality and were developed using MBSwE.

Since auto-generated code (AGC) can strongly influence the software metrics (e.g., lines of code), which can result in misleading observations, we considered two versions of the "R2U2" app, one with auto-generated code (R2U2 with AGC) and another R2U2 without auto-generated code (annotated simply as R2U2).

V. DATA EXTRACTION AND METRIC DEFINITION

We had access to the AOS's source code, commit logs, and the bug tracking system as all artifacts were stored in a Git repository. Throughout the development, engineering tools like Jira (for issue tracking) and bamboo (combined with a customized scenario testing) were used. All project documentation and communication between researchers was kept in Confluence; however there was no strict development protocol or process.

For this study, we extracted data from the Git repository as it provided all information about the history of the project, as well as some documentation. In addition, we conducted semi-structured interviews with several developers and researchers of the AOS project and utilized the relevant AOS publications [13]–[15].

TABLE II
EXTRACTED METRICS AND THEIR DESCRIPTIONS

Metric	**Description**
Lines of code	Number total lines of code (LoC) in a file
Commits	Number of times a file was edited
LoC added	Sum of all LoC added to a file for all commits
LoC removed	Sum of all LoC removed from a file for all commits
Codechurn	Sum of LoC added and LoC removed for all commits
Bugfixes	Number of times a file was involved in bug fixing, for all commits

Our analysis was focused on the following file types: h, c, cc, cpp, sh, hpp, ple, plp, hh, plx, prolog, json, and txt. (plp, ple, and plx files contain domain-specific Plexil planning language.) We chose to analyze files with these file extensions because they contain code related to the development of AOS. Note that we only included the Cmakelists.txt files in the analysis. Other files with txt extension (e.g., README and dev-notes) were not considered. In total, 983 files belonging to the 26 different apps were analyzed.

For each file in each of the 26 apps, we first extracted the metrics listed in Table II at file level, and then aggregated them at app level.

We used a Python script to determine the size of each file, measured in LoC. The modifications to a file during the course of its existence were captured by the change metrics. We extracted the same change metrics as previous studies on software fault proneness (e.g., [22], [23]). PyDriller [24] was used to facilitate extracting the change metrics from the commit logs.

The development team of AOS did not track the software bugs consistently in the Jira issue tracking system. Due to the small number of bugs entered in Jira, we used the version control logs to identify commits that included changes related to fixing software bugs. For this purpose, following the related work (e.g., [22]), we mined through the textual descriptions of all commits for a set of regular expressions that contained words such as “bug”, “fix” and “fault”. If found, the corresponding commit was labeled as a “bugfix”. For the analysis at app (i.e., component) level, if a commit with bugfixes led to changes in only one app, even if several files in that app were changed, we counted it as one bugfix at app level. If a commit with bugfixes led to changes in multiple apps, we counted it as one bugfix for each app. Files that had at least one bugfix commit were marked as faulty files.

VI. Empirical Results

In this section we discuss the empirical results as they pertain to our research questions RQ1 – RQ5.

A. *Size of AOS apps*

When looking at the static code size (LoC) for each category of apps, MBSwE apps tended to be significantly larger than all the other apps (Fig. 2). The reason simply being that the code generator used to produce C code from Matlab included the source code of numerous libary functions, thus resulting in an extremely large source code. For all other app categories, the size of the source code seemed to be comparable. The distribution of LoC for the autonomous components is larger than for the rest. In most cases the different component sizes could be attributed to app-specific artifacts like plans, failure models or operational concepts, which are needed to support the autonomous functionality.

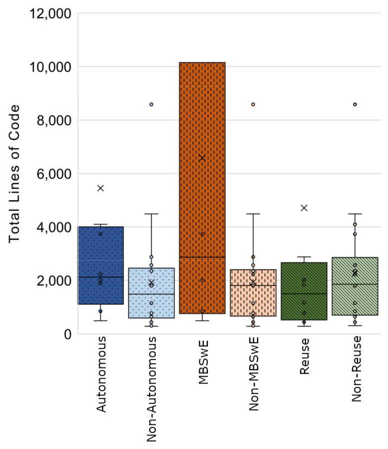

Fig. 2. Box plot of the app's size in LoC, for each app category. The outliers are for Autonomous: Plexil; Non-Autonomous: cf; MBSwE: Plexil; Non-MBSwE: cf; Reuse: Plexil; Non-Reuse: cf. Plexil is not shown because as an outlier it makes viewing the medians and IQR difficult.

B. *Changes in Code: Number of Commits and Code Churn*

For a software development project, not only the code sizes, as discussed in subsection VI-A are of importance, but also how often and to what extent the code was changed during the development. Here, the relevant metrics are the number of commits and the code-churn (i.e., sum of LoC added and LoC removed).

As shown in Fig. 3, only two of the apps, namely the planner Plexil and the prolog navigation app had extremely large numbers of commits, compared to the rest of the apps. Whereas the large number of customized Plexil plans, which had to be adapted numerous times during development and testing, might have contributed to the high number of commits, the situation for the prolog app is not very clear. The frequency with which the developers do commits during development might be a driving (if not the major) factor for these numbers. Therefore, these numbers may not be very indicative. The inset in Fig. 3 shows a blow-up. It can be seen that the autonomous apps, which are in the center of attention in the AOS project, as well as the supporting apps (e.g., mav_bridge) had a substantially higher number of commits than the more OS-related ones.

To account for the size of the apps, we also explored the number of commits per KLoC. When normalized by size, Plexil is no longer the app with the most commits. Instead, Prolog has the most number of commits per KLoC, followed by plex_mav, plex_nlp, and aos_lib.

Interestingly, the code churn of the files in autonomous apps was particularly high (see Fig. 4). This is because research and algorithms development took place. Code churn for MBSwE apps was also relatively high because automatic code generation tends to produce the entire code from scratch for each new version. Unsurprisingly, the apps with reuse had much smaller code churn than apps with no reuse.

C. *Fault proneness of apps: Quantitative analysis*

In this section we study the number of bugs that were identified and fixed during the AOS development. In traditional sense, all AOS bugs are pre-release bugs. As discussed in Section V, we use the term “Bugfix” to refer to a commit that involved fixing one or more software bugs. In this section, we present quantitative analysis of AOS fault proneness focused on number of Bugfixes and Bugfixes per KLoC for each AOS app, as well as per app category. The bar graph of the total number of Bugfixes for each AOS app is shown in Fig. 5. Plexil had a total of 457 commits involving bugfixes, which is significantly higher than any other AOS app. The app with the second largest amount of Bugfixes was prolog with 74 bugfixes. To better observe the trend, Fig. 5 has an inset bar graph, with a smaller range on y-axis.

To account for the size, we examined the number of Bugfixes per KLoC, which are shown in Fig. 6. As in case of number of commits and codechurn, when normalized by size, Plexil is no longer the most fault prone app. Instead, prolog app, which had 33 bugfixes per KLoC, was the most fault prone.

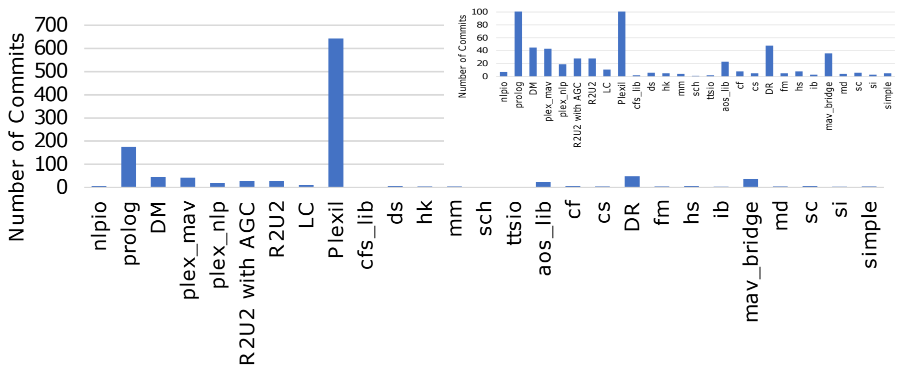

Fig. 3. Bar graph of the total number of commits for each app. The inset shows a blow up with bars for prolog and plexil cut off.

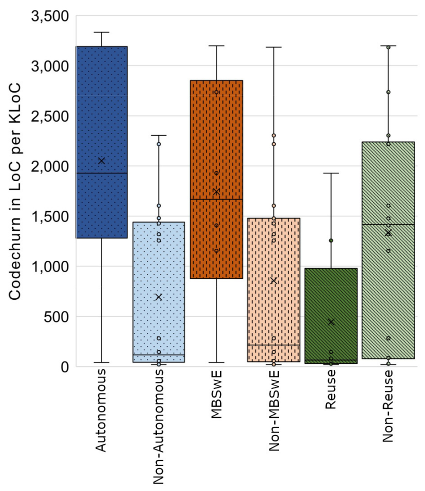

Fig. 4. Box plots of the total Codechurn (in LoC) per KLoC, for each app category

Fig. 7 shows the box plots of the number of Bugfixes per KLoC for the three app categories. The Autonomous apps had significantly higher median number of bugfixes per KLoC than the Non-autonomous (i.e., 10.54 compared to 3.64). The results for MBSwE apps were similar, which is due to the large overlap of these two categories. AOS apps with reuse had a median number of 3.40 Bugfixes per KLoC, which as expected is much smaller than the apps with no reuse that had 6.19 median number of bugfixes per KLoC.

D. *Distribution of Bugfixes among app's files*

In this subsection we explore the location of Bugfixes at file level for six selected AOS apps: prolog, DM, R2U2, Plexil, DR, and mav_bridge. The first four are autonomous apps, while the last two are non-autonomous apps.

prolog, DM, R2U2, and Plexil apps had zero median number of Bugfixes at file level, while DR and mav_bridge had median number of one Bugfix. Plexil had the highest number of outliers, i.e., files with larger number of Bugfixes. Prolog and R2U2 had the lowest variation (IQR=0), followed by DM, Plexil, and DR (IRQ=1), and mav_bridge, which had the highest variation (IQR=2). When normalized by LoC, as can be seen in Fig. 8, prolog and R2U2 still have the lowest IQR and Plexil has the largest number of outliers. DR had the highest median Bugfixes of 0.02 per LoC and the highest IQR of 0.07.

Fig. 9 presents the histograms for the number of bugfixes per file, for each selected app. Note that x-axes in Fig. 9 have the same range (from 0 to 24 Bugfixes), while y-axes have different ranges because apps have different sizes in number of files. As evident from these histograms, all apps have skewed distributions of number of Bugfixes per file, with the highest number of files having zero Bugfixes. The only exception is the DR app, which has the highest number of files with one Bugfix, followed by the number of files with zero Bugfixes. Plexil had the most skewed distribution, with the largest number of outliers (i.e., files with multiple Bugfixes, such as one file with 11 Bugfixes, one file with 15 Bugfixes, two files with 16 Bugfixes each, one file with 18 Bugfixes, and one file with 22 Bugfixes).

To further explore the uneven distribution of software bugs, we compiled Table III which presents the total number of files in each app, the percentage of Bugfixes in the closest to 20% of the most fault prone files, as well as the percentage of files that contain 100% of Bugfixes. The results show that prolog had 100% of Bugfixes in only 17% of the files. Similarly, R2U2 had 91% of all Bugfixes in 19.6% of the files, and all Bugfixes in only 21.6% of the files. DM and Plexil had 50% and 51% of the Bugfixes in 18.5% and 19.8% of the files, respectively. These two apps had 100% of Bugfixes in only 37.0% and 38.5% of the files, respectively. These results are consistent with the Pareto principle of uneven distribution of Bugfixes across files, and indicate that testing these parts of software provides cost effective way to improve software reliability.

The two non-autonomous applications, DR and mav_bridge, have least Bugfixes in the 20% most fault prone files (i.e., 27% and 33%, respectively). 100% of their Bugfixes were located in 73.3% and 60.0% of their files, respectively. This can be explained by the facts that DR and mav_bridge apps had proportionally smaller number of files with zero Bugfixes and more evenly distributed Bugfixes.

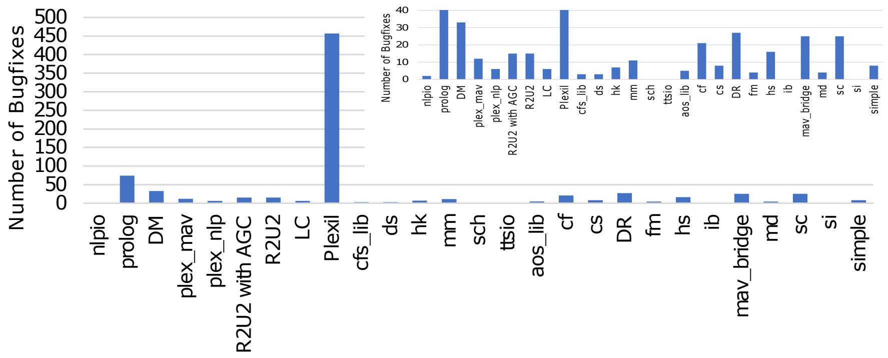

Fig. 5. Bar graph of the total number of Bugfixes for each AOS app. The inset shows a blow up with bars for prolog and plexil cut off.

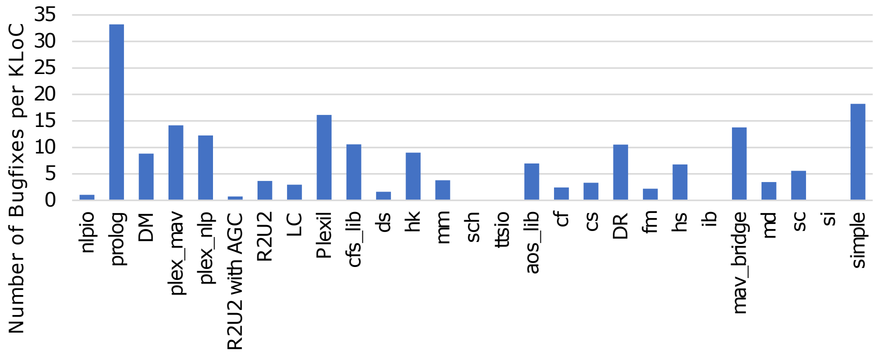

Fig. 6. Bar graph of the number of Bugfixes per KLoC, for each AOS app

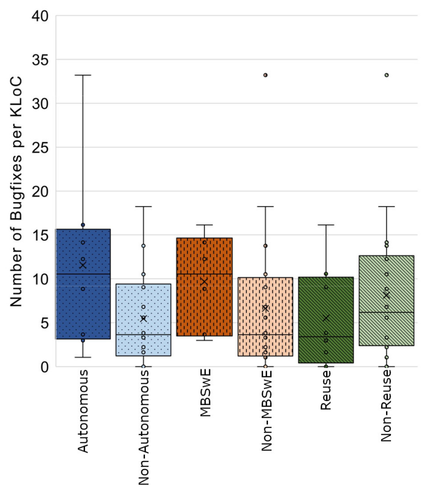

Fig. 7. Box plots of the number of Bugfixes per KLoC, per app category. Outliers are Non-MBSwE: prolog; Non-Reuse: prolog.

E. Characteristics of bugs: Qualitative analysis

As discussed in subsection VI-D, for the selected apps, only a few files carry the majority of bugs. A closer look at the individual files and the characteristics of the bugfixes revealed the following for the individual apps.

prolog app was developed as an interface app between AOS and programs written in prolog. It was developed from scratch. Most bugs and fixes occurred in the configuration part of the app (CMakelist and AOS-specific header files). Although the prolog app was not large (around 2,200 LoC), it had the second highest number of bugfixes (i.e., 74) and the highest number of bugfixes per KLoC (i.e., 33).

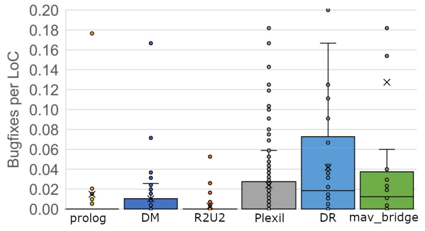

Fig. 8. Boxplot of the number of Bugfixes per LoC for the six selected apps. The y-axis is rescaled from 2.5 to 0.2 Bugfixes per LoC to make the medians and IQR visable.

Autonomous Decision Maker (DM) app was developed from scratch and was, in large parts, implemented in Prolog. Bugs were found and fixed in the AOS/prolog interface, consistency of messages, and changes in the routine for performing low-level active diagnostics. Since this part directly interfaces to mav_bridge, several of the same bugs were encountered. Bugs in the prolog code included wrong number/type of parameters, initialization of the knowledge base, and removal of unnecessary backtracking points (cuts).

R2U2 had the most bugs in its main program. They concerned handling cFS messages and interfacing to the R2U2 engine. Most bugs included configuration inconsistencies and memory-related issues. In the R2U2 engine, the bugs con-

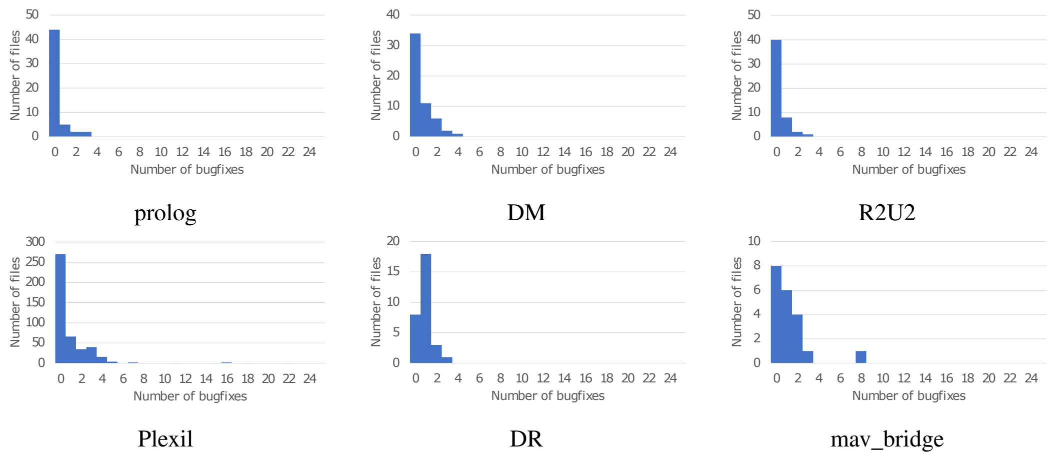

Fig. 9. Histograms of the number of Bugfixes per file for selected apps. The maximum number of Bugfixes per file are 3 for prolog, R2U2, and DR, 4 for DM, 8 for mav_bridge, and 22 for Plexil.

TABLE III
Summary statistics of the six selected apps

	App [Category]					
	prolog [A]	DM [A/M]	R2U2 [A/M]	Plexil [A/M/R]	DR	mav_bride
Total number of files	53	54	51	439	30	20
% of most fault prone files	17.0%	18.5%	19.6%	19.8%	20.0%	20.0%
% of Bugfixes in these files	100%	50%	91%	51%	27%	33%
% of files with 100% of Bugfixes	17.0%	37.0%	21.6%	38.5%	73.3%	60.0%

cerned proper initialization of the engine, index/counter errors, and algorithm inconsistencies. Not surprisingly, both variants of R2U2 (R2U2 with AGC and R2U2 without AGC) contained the same number of bugs, as there were no bugs detected in the AGC code. R2U2 (without AGC) had the lowest number of Bugfixes per KLoC (only 4.29) and 100% of Bugfixes were located in only 21.6% of the files.

Plexil app comprises the on-board planning system. It therefore gets sensor and status information about the aircraft from AOS, and issues commands to the AOS software bus. During its iterative development, numerous additions and fixes needed to be applied to the files containing the interface code. Several C++ files were affected. The main “flight execution plan”, which is one of the core Plexil plans used in AOS, had 16 Bugfixes. These bugs were of various kinds and occurred during its development and extension of capabilities. Plexil is the largest app (with close to 29,000 LoC) and has the highest number of Bugfixes (i.e., 457), as well as very high number of Bugfixes per KLoC (i.e., 15.71). This result is consistent with recent related works [9], [10] which also reported that the planning component was among the most fault prone. All of Plaxil’s Bugfixes were located in around 38.5% of the files (see Table III).

Diagnostic Reasoner (DR) app was developed from scratch to implement a diagnostic reachability-based reasoning engine. As expected, bugfixes showed up in the main program, the reasoning algorithm, as well as in the unit-test programs. Only 27% of Bugfixes were located in the 20% most fault prone files. DR Bugfixes were widely distributed across 73.3% of its files.

mav_bridge app provides the interface between AOS and the low-level ArduPilot autopilot. Flight commands are converted into the MAV format before transmission, and sensor data are read from the autopilot and converted into cFS messages. The development of this app was done iteratively as additional requirements for new commands had to be implemented over time. Furthermore, it turned out that specific AOS commands could not be directly mapped to ArduPilot MAV commands, requiring a large amount of trial-and-error. The result of this development is reflected in numerous bugfixes in the app’s main code file, as well as the corresponding header files, containing details of the interface. mav_bridge has the least amount of LoC of the six selected apps (i.e., around 1,700) and had very high number of Bugfixes per KLoC (i.e., 14.76). The Bugfixes made in mav_bridge were widely distributed; 100% of Bugfixes were located in 60% of the files.

VII. Threats to Validity

In this section, we describe several threats to the validity of this study and the measures taken to mitigate them.

Construct validity addresses whether we are testing what we intended to test. We conducted semi-structured interviews with multiple members of the software development and research team to understand the design and implementation of AOS and different sources that can be used to collect the data of interest. Since AOS was developed as research project, the development did not follow a specific process or coding guidelines. For example, only some developers used the issue tracking system Jira to report software bugs. To overcome this

TABLE IV
SUMMARY OF THE MAIN FINDINGS

RQ1: What are the typical sizes of AOS apps?	
	MBSwE apps were the largest in size (LoC), with the largest variability. The two largest apps had over 20,000 LoC; most apps had from several hundred to several thousand LoC.
RQ2: Are there differences in change proneness (measured in number of commits and code churn) among apps from different categories (i.e., Autonomous, MBSwE, Reused)?	
	Autonomous apps had significantly higher median number of commits per KLoC than Non-autonomous apps (i.e., 17.39 compared to 4.00 commits per KLoC). The MBSwE apps had same median number of commits per KLoC as Autonomous apps, which was mainly due to the fact that these two categories share seven apps. The apps with reuse were least change prone, with only around 5 commits per KLoC and the smallest IQR.
	Autonomous apps had significantly higher median codechurn per LoC than Non-autonomous apps (i.e., 1,703 compared to 116 LoC per KLoC). The MBSwE apps had slightly lower median codechurn per KLoC of 1,667 than the Autonomous app. As expected, AOS apps with reuse had least codechurn per KLoC, with significantly lower median of 65 LoC per KLoC.
RQ3: Are there differences in fault proneness among apps from different categories (i.e., Autonomous, MBSwE, Reused)?	
	Autonomous apps had significantly higher median number of bugfixes per KLoC than the Non-autonomous (i.e., 10.54 compared to 3.64). The results for MBSwE apps were similar, which is due to the large overlap of these two categories. AOS apps with reuse had 3.40 bugfixes per KLoC, which is much smaller than the apps with no reuse that had 6.19 median number of bugfixes per KLoC.
RQ4: Are the locations of Bugfixes unevenly distributed among app's files?	
	For the six selected apps analyzed at file level, the number of Bugfixes at file level followed skewed distributions. While the largest number of files had zero Bugfixes, there were files with multiple Bugfixes (up to 22 Bugfixes for one file in Plexil app).
	Two of the six apps, prolog and R2U2, had 100% of bugfixes in 17% and 21.6% of the files. DM and Plexil had 50% and 51% of the Bugfixes in slightly less than 20% of the files. However, they had 100% of the Bugfixes in 37% and 38.5% of the files respectively. The two non-autonomous apps, DR and mav_bridge, had the least amount of Bugfixes in the 20% most fault prone files (i.e., 27% and 33%, respectively). 100% of their bugs were located in 73.3% and 60% of the files respectively.
RQ5: What are the characteristics of software bugs in different AOS apps?	
	Autonomous components were more fault prone than Non-autonomous components, likely due to larger models and plans being part of such components. Although in category autonomous, numerous bugfixes occurred in the "traditional" parts of an autonomous app (like interface code) rather than the core functionality.

limitation, we used data triangulation and utilized the commit logs to identify the bugfixing commits. To account for apps and files of vastly different sizes, in addition to the raw numbers of commits, code churn, and bugfixes, we also studied the values normalized by size (in KLoC). In general, some (small) preparation phase during a project setup and a few process guidelines could be extremely helpful for later analysis and would further mitigate some threats to validity.

Internal validity concerns influences that can affect the variables and metrics without researchers' knowledge. One typical threat to internal validity is data quality. As in any other study of software fault proneness, potential lack of information in commits' comments may have led to underestimating the number of software bugs / bugfixes.

Conclusion validity concerns the ability to draw correct conclusions. This study is based on analysis of pre-release software bugs and the results may not translate to post-release bugs. It should be noted that none of the related works on empirical characterization of fault proneness of autonomous systems [9]–[12] specified if they analyzed pre-release bugs, post-release bugs, or both.

External validity concerns the generalizability of results. Even though this work is based on a real-life large case study of an autonomous system, we cannot claim that the results based on one case study would be valid for other autonomous systems. For example, the iterative development process used for AOS, with several apps being implemented in a later stage and in different versions, could lead to results that cannot be easily carried over to a more industrial software development.

VIII. Summary of the Main Results and Conclusions

In this paper, we performed an in-depth empirical analysis of a complex autonomous software system AOS, which also used model-based software engineering and software reuse. Based on the available artifacts in a Git repository and semi-structured interviews with AOS developers, we explored five research questions focused on the size of software components, changes to the code during development, fault proneness, the spread of software bugs both at component and file level, and the characteristics of software bugs. Table IV summarizes our main findings.

Several of our evidence-based findings are important for researchers and practitioners alike. Thus, our finding that the autonomous components are significantly more prone to changes and bugs compared to non-autonomous components indicates the challenges associated with the development of autonomous functionality and necessities further improvements. Another result of our analysis showed that the majority of bugs are located in a small number of components, and within each component in a small number of files. These findings show once again that allocating more development and testing effort to these parts of a software system provides cost effective way to improve its reliability, safety, and security. Last but not least, the qualitative analysis of software bugs in individual components provides lessons learned that can be used to avoid similar bugs in future autonomous systems.

Acknowledgements

This work was funded in part by the NASA Software Assurance Research Program (SARP). The authors would like to thank the colleagues from the NASA Ames Research Center who provided the access to the AOS artifacts and shared the relevant information with us.

References

[1] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Trans. Software Engineering*, vol. 35, no. 4, pp. 484–496, 2009.

[2] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 178–187.

[3] M. Hamill and K. Goseva-Popstojanova, "Exploring the missing link: An empirical study of software fixes," *Software Testing, Verification and Reliability Journal*, vol. 24, no. 5, pp. 49–71, 2013.

[4] ——, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015.

[5] A. Di Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 509–519.

[6] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2012, pp. 271–280.

[7] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[8] K. Goseva-Popstojanova, T. Kyanko, and N. Nkwocha, "Benefits and challenges of model-based software engineering: Lessons learned based on qualitative and quantitative findings," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 413–423.

[9] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, "A comprehensive study of autonomous vehicle bugs," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020, pp. 385–396.

[10] S. Tang, Z. Zhang, J. Tang, L. Ma, and Y. Xue, "Issue categorization and analysis of an open-source driving assistant system," in *International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2021, pp. 148–153.

[11] M. Taylor, J. Boubin, H. Chen, C. Stewart, and F. Qin, "A study on software bugs in unmanned aircraft systems," in *International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2021, pp. 1439–1448.

[19] D. McComas, "NASA/GSFC's Flight Software Core Flight System," in *Flight Software Workshop*, 2012.

[12] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui, "An exploratory study of autopilot software bugs in unmanned aerial vehicles," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 20–31.

[13] M. Lowry, A. R. Bajwa, T. Pressburger, A. Sweet, M. Dalal, C. Fry, J. Schumann, D. Dahl, G. Karsa, and N. Mahadevan, "Design Considerations for a Variable Autonomy Executive for UAS in the NAS," in *AIAA Information Systems-AIAA Infotech @ Aerospace*, 2018.

[14] M. Lowry, T. Pressburger, D. Dahl, and M. Dalal, "Towards Autonomous Piloting: Communicating with Air Traffic Control," in *AIAA Scitech Forum*, 2019.

[15] J. Schumann, N. Mahadevan, A. Sweet, A. R. Bajwa, M. Lowry, and G. Karsai, *Model-based System Health Management and Contingency Planning for Autonomous UAS*, 2019. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2019-1961

[16] SAE On-Road Automated Vehicle Standards Committee, "Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems," *SAE Standard J*, vol. 3016, no. 1, pp. 1–16, 2014.

[17] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 331–342.

[18] F. Zampetti, R. Kapur, M. Di Penta, and S. Panichella, "An empirical characterization of software bugs in open-source cyber–physical systems," *Journal of Systems and Software*, vol. 192, p. 111425, 2022.

[20] V. Verma, A. Jonsson, C. Pasareanu, and M. Iatauro, "Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations," in *Spacecraft Control and Operations, American Institute of Aeronautics and Astronautics Space 2006 Conference*, 2006.

[21] Federal Aviation Administration. (2020) Unmanned Aircraft System (UAS) Traffic Management (UTM): Concept of Operations V2.0. [Online]. Available: https://www.faa.gov/uas/research_development/traffic_management/media/UTM_ConOps_v2.pdf

[22] S. Krishnan, C. Strasburg, R. R. Lutz, K. Goseva-Popstojanova, and K. S. Dorman, "Predicting failure-proneness in an evolving software product line," *Information and Software Technology*, vol. 55, no. 8, pp. 1479–1495, 2013.

[23] T. Devine, K. Goseva-Popstajanova, S. Krishnan, and R. R. Lutz, "Assessment and cross-product prediction of spl quality: accounting for reuse across products, over multiple releases," *Automated Software Engineering*, vol. 23, pp. 253–302, 2016.

[24] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 908–911.